

Assignment no.4

- **Write a program on Knapsack problem using Greedy method**

Code:-

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.ratio = value / weight

def fractional_knapsack(capacity, items):
    # Sort items by profit/weight ratio in
    descending order
    items.sort(key=lambda x: x.ratio,
reverse=True)

    total_value = 0.0
    selected_items = []

    for item in items:
        if capacity == 0:
            break

        # If item can be taken completely
        if item.weight <= capacity:
            capacity -= item.weight
            total_value += item.value
```

```

        selected_items.append((item.value,
item.weight, 1)) # 1 means fully taken
    else:
        # Take fractional part
        fraction = capacity / item.weight
        total_value += item.value * fraction
        selected_items.append((item.value,
item.weight, fraction))
        capacity = 0

    return total_value, selected_items

```

----- Example -----

```

if __name__ == "__main__":
    capacity = 50

    items = [
        Item(60, 10),
        Item(100, 20),
        Item(120, 30)
    ]

    max_value, selected =
fractional_knapsack(capacity, items)

    print("Maximum value obtained:", max_value)
    print("\nItems selected (value, weight, fraction
taken):")

```

```
for item in selected:  
    print(item)
```

Assignment no.3

- **Write a program to solve job sequencing using deadline using Divide and Conquer approach**

Code:-

```
class Job:  
    def __init__(self, job_id, deadline, profit):  
        self.id = job_id  
        self.deadline = deadline  
        self.profit = profit  
  
def merge_jobs(left_jobs, right_jobs):  
    """  
    Merge two job lists based on descending profit.  
    """  
    return sorted(left_jobs + right_jobs, key=lambda  
x: x.profit, reverse=True)  
  
def schedule_jobs(jobs):  
    """  
    Schedule jobs to maximize profit using greedy  
    slot allocation.  
    """  
    max_deadline = max(job.deadline for job in jobs)  
    slots = [None] * max_deadline
```

```
total_profit = 0
```

```
for job in jobs:
```

```
    # Try to schedule job at the latest available slot
```

```
    for j in range(min(max_deadline, job.deadline) - 1, -1, -1):
```

```
        if slots[j] is None:
```

```
            slots[j] = job
```

```
            total_profit += job.profit
```

```
            break
```

```
scheduled_jobs = [job.id for job in slots if job is not None]
```

```
return scheduled_jobs, total_profit
```

```
def job_sequencing_divide_conquer(jobs):
```

```
    """
```

```
    Divide and Conquer Job Sequencing.
```

```
    """
```

```
    # Base case
```

```
    if len(jobs) <= 1:
```

```
        return jobs
```

```
    # Divide
```

```
    mid = len(jobs) // 2
```

```
    left = job_sequencing_divide_conquer(jobs[:mid])
```

```
    right =
```

```
    job_sequencing_divide_conquer(jobs[mid:])
```

```
# Conquer (Merge step)
merged = merge_jobs(left, right)
```

```
return merged
```

```
# ----- Example -----
```

```
if __name__ == "__main__":
    jobs = [
        Job('J1', 2, 100),
        Job('J2', 1, 19),
        Job('J3', 2, 27),
        Job('J4', 1, 25),
        Job('J5', 3, 15)
    ]
```

```
# Step 1: Sort using Divide & Conquer
sorted_jobs =
    job_sequencing_divide_conquer(jobs)
```

```
# Step 2: Schedule jobs greedily
scheduled, profit = schedule_jobs(sorted_jobs)
```

```
print("Scheduled Jobs:", scheduled)
print("Total Profit:", profit)
```

Assignment 2.b

Title: Write a Program to sort an array of N elements using Quick sort.

Code:-

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Driver code
arr = []
n = int(input("Enter the number of elements in the array: "))
for i in range(n):
    arr.append(int(input(f"Enter element {i + 1}: ")))
```

```
quick_sort(arr, 0, n - 1)  
print("Sorted array:", arr)
```

Assignment 2.a

Title: Write a program that uses both recursive and non-recursive functions to perform the following searching operations for a key value in a given list of integers:

Code:-

```
# Non-recursive Linear Search
def linear_search(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1

# Recursive Linear Search
def recursive_search(arr, key, index=0):
    if index >= len(arr):
        return -1
    if arr[index] == key:
        return index
    return recursive_search(arr, key, index + 1)

# Driver code
arr = [10, 25, 30, 45, 50]
key = int(input("Enter the element to search: "))
result1 = linear_search(arr, key)
result2 = recursive_search(arr, key)
if result1 != -1:
    print("Non-Recursive: Element found at index", result1)
else:
    print("Non-Recursive: Element not found")
if result2 != -1:
    print("Recursive: Element found at index", result2)
```


else:

print("Recursive: Element not found")

Assignment no.1 B

program to sort the elements of array using recursive Heap sort

code:-

```
# Recursive function to heapify a subtree rooted  
at index i
```

```
def heapify(arr, n, i):
```

```
    largest = i
```

```
    left = 2 * i + 1
```

```
    right = 2 * i + 2
```

```
    # Check if left child exists and is greater than  
    root
```

```
    if left < n and arr[left] > arr[largest]:
```

```
        largest = left
```

```
    # Check if right child exists and is greater  
    than largest so far
```

```
    if right < n and arr[right] > arr[largest]:
```

```
        largest = right
```

```
    # If largest is not root
```

```
    if largest != i:
```

```

    # Swap
    arr[i], arr[largest] = arr[largest], arr[i]

    # Recursively heapify the affected subtree
    heapify(arr, n, largest)

# Recursive heap sort function
def heap_sort(arr, n=None):
    if n is None:
        n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    if n <= 1:
        return

    # Move current root (largest) to the end
    arr[0], arr[n - 1] = arr[n - 1], arr[0]

    # Recursively sort the remaining heap
    heap_sort(arr, n - 1)

# Example usage

```

```
arr = [12, 11, 13, 5, 6, 7]
print("Original array:")
print(arr)
heap_sort(arr)
print("Sorted array:")
print(arr)
```

Assignment no.5 Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity

```
class Edge:
```

```
    def __init__(self, u, v, w):
        self.u = u  # source vertex
        self.v = v  # destination vertex
        self.w = w  # weight
```

```
def bellman_ford(vertices, edges, source):
```

```
    # Step 1: Initialize distances
```

```
    dist = [float("inf")] * vertices
```

```
    dist[source] = 0
```

```
    # Step 2: Relax edges (V-1 times)
```

```
    for _ in range(vertices - 1):
```

```
        for edge in edges:
```

```
            if dist[edge.u] != float("inf") and dist[edge.u] + edge.w < dist[edge.v]:
```

```
                dist[edge.v] = dist[edge.u] + edge.w
```

```
    # Step 3: Check negative cycle
```

```
    for edge in edges:
```

```
        if dist[edge.u] != float("inf") and dist[edge.u] + edge.w < dist[edge.v]:
```

```
            print("Graph contains a negative weight cycle!")
```

```
            return None
```

```
    return dist
```

```
# Example
```

```
edges = [  
    Edge(0, 1, 1),  
    Edge(0, 2, 4),  
    Edge(1, 2, -3),  
    Edge(1, 3, 2),  
    Edge(2, 3, 3)  
]
```

```
vertices = 4
```

```
source = 0
```

```
distances = bellman_ford(vertices, edges, source)
```

```
if distances is not None:
```

```
    print("Shortest distances from source:", distances)
```

Assignment no.7 Write a recursive program to find the solution of placing n queens on the chessboard so that no two queens attack each other using

```
def print_board(board, n):  
    for i in range(n):  
        for j in range(n):  
            print(board[i][j], end=" ")  
        print()
```

```
def is_safe(board, row, col, n):
```

```
    # Check column  
    for i in range(row):  
        if board[i][col] == 1:  
            return False
```

```
    # Check left diagonal  
    i, j = row - 1, col - 1  
    while i >= 0 and j >= 0:  
        if board[i][j] == 1:  
            return False  
        i -= 1  
        j -= 1
```

```
    # Check right diagonal  
    i, j = row - 1, col + 1  
    while i >= 0 and j < n:  
        if board[i][j] == 1:  
            return False  
        i -= 1
```

```
j += 1
```

```
return True
```

```
def solve(board, row, n):
```

```
    # Base case
```

```
    if row == n:
```

```
        return True
```

```
    for col in range(n):
```

```
        if is_safe(board, row, col, n):
```

```
            board[row][col] = 1  # Place queen
```

```
            if solve(board, row + 1, n):
```

```
                return True
```

```
            board[row][col] = 0  # Backtrack
```

```
    return False
```

```
# Main
```

```
n = 4
```

```
board = [[0]*n for _ in range(n)]
```

```
if solve(board, 0, n):
```

```
    print("Solution exists:\n")
```

```
    print_board(board, n)
```

```
else:
```

```
    print("No solution exists")
```